Trinity College Dublin
The University of Dublin

School of Computer Science and Statistics
CS1021 - Introduction to Computing I
Conor Gildea - [redacted]
Assignment #1

Assignment #1 Simple Statistics Calculator

1. Console Input

Basic Functionality

Due to the nature of ARM assembly being a low level programming language, it makes it slightly more difficult to gather console input and be able to interpret it, than the likes of a higher level language such as Java. For Stage 1, my program first prompts the user with a message to input their number value. Due to the limitations of ARM assembly, we must process each digit separately and do some light calculations to make the value match what the user meant to enter. To read the user's input we must complete it inside a loop, which we shall name "read". Our first major problem is the conversion from ASCII symbol hexadecimal values (what the user typed) to the hexadecimal values the numbers are meant to represent.

E.G. For "406" each digit must first be treated individually as their hexadecimal values.

"4" = 0x34 "2" = 0x32 "6"=0x36

The numbers currently are equal to their ASCII symbol values. We must now minus 0x30 from each value to convert it from ASCII symbol values to the actual value the user intended.

0x34 – 0x30 = 0x04 = 4 0x32 – 0x30 = 0x02 = 2 0x36 – 0x30 = 0x06 = 6

We must now develop a system to make: 4, 2, 6 be equal to 426.
To do this we must first introduce a running total variable (runningTotal) with a value of zero and a constant with a value of 10 (which we shall refer to as simply "TEN"). After the first digit is entered we then let that be added on to the running total variable.

runningTotal = runningTotal + 4

After the second digit is entered, we must now multiply the previous running total by the second variable we declared with the value of 10.

runningTotal = runningTotal * TEN //runningTotal = 40

At the end of the reading loop we then must again add on the new digit to the running total.

runningTotal = runningTotal + 2 //runningTotal = 42= 0x2A after "4" and "2" entered

After this process is completed another time, after the third digit "6" is entered, the value should be correctly processed as "4", "2", "6" = 426 = 0x1AA.

Now we are able to have the user input a value and have the values be interpreted properly however we must now allow the user to specify when one value is completed so they can enter another. To do this we check has the user entered the hexadecimal value for “Enter”, if so we can cause the program to branch out from our “read” loop. Outside of our “read” loop we should contain another overall loop, which we shall refer to as simply “loop”. This will allow the user to re-enter the “read” loop to enter more values. However we shall halt this process and continue it in stage 2, as for stage 1 the last value must be saved in R4 and we must overwrite the last value to enter multiple values.

Negative Numbers Processing – Extra Mile

For the “extra mile” in this stage, I enabled by program to accept positive and negative values and I also prevented invalid input from being entered. Due to ARM Assembly’s nature we must program to compute negative numbers ourselves, unlike higher level languages. To allow my program to accept both positive and negative numbers I had to make a few adjustments to my program. The basic summary of my adjustments were that if (“-“) as an ASCII symbol was entered, it would then branch to the end of the “read” loop, skipping the calculations however it would also set a boolean “negativeValue” to true (R12 = 0x01). The user would then enter the first digit of their negative value and it would be processed as normal. After the value was finished being entered, if (negativeValue == true), the program would then convert the negative number into its 2’ Compliments counterpart. I then used code based on the following pseudocode to convert the negative numbers into their 2’ Compliments counterpart.

```
if (negativeNumber=true)      //Converts negative numbers to 2 Compliment Form
{
        value = NOT value (invert bits)
        value = value + 1 (add 1)
}
```

Invalid Input Handling – Extra Mile

Lastly for preventing invalid input I simply had a series of almost if statements that would only allow a certain range of ASCII symbols to be entered, with the exceptions of (“-“, ”.” and ”Enter”). To do this I processed the exceptions first then I processed the other cases:

```
If (userInput >= 0x30)&& (userInput<=0x39)                //0x30 = “0”, 0x39 = “9”
{
        …
}
```

If the userInput was none of the exceptions and outside of the above range, my program then prints a message saying “Invalid Input. Please restart and try again and enter valid input”. This prevents my program from trying to process invalid input. E.G. Trying to process the hexadecimal value for a letter symbol "m", when it should only process number symbols.

Methodology for Stage One – Console Input

Input: Followed by "Enter" and "."	**Input: Hexadecimal Values**	**Output: Applicable Registry Value**	**Console Output After Welcome**	**Correct/Wrong? Why?**
"0"	0x30	0x00	N/A – Stage 1	Correct – The value "0" was saved as 0.
"230"	0x32,0x33,0x30	0xE6	N/A – Stage 1	Correct – The hexademical value of "230" is 0xE6.
"-230"	0x2D,0x32, 0x33,0x30	0xFFFFFF1A	N/A – Stage 1	Correct – The hexademical value of "-230" in 2 Compliment form is 0xFFFFFF1A.
"m"	0x6D	0x00000000 (No Change)	"Invalid Input, please restart and enter valid input"	Correct – The program handled the case of a non-number correctly.
No Input	None	0x00000000 (No Change)	N/A – Stage 1	Correct – The program treated no input followed by the "enter" and "." key as if it was zero, this is the quite reasonable handling. And would be more than suitable given the user was meant to be "well behaved".

2. Sample Sequence and Statistic Computation

Stage Two of the assignment was quite non-trivial. In this stage my program had to calculate the following listed five statistical measures. For my extra mile in this section I also calculated the range, another element of my extra mile in this section was correctly handling the negative numbers to allow them to be displayed properly.

Count

Calculating the count was the easiest statistical measure to calculate. It would just increase the count by 1 every time a value was finished being entered.

```
count++;
```

Sum

Calculating the sum was also quite non trival. As I had already converted negative numbers into their correct 2 Compliment form, this allowed me to simply add on the newest value to the count and it processed correctly each time.

```
sum = sum + valueEntered;
```

Maximum and Minimum

As the maximum and minimum both calculate so similarly but in opposite directions, I shall treat them both together. This calculation first consisted of setting the first value entered when the count was =1, to be both the maximum and the minimum.

```
If (count==1)

{

    maximumNumber = valueEntered;

    minimumNumber = valueEntered;

}
```

The next computation to calculate the Max and the Min was to simply check was the value entered bigger than the current maximum or smaller than the current minimum. As I was allowing negative numbers to be inputted this looked like it would have been quite difficult in ARM Assembly, however with the use of the signed value conditional branches, it made this computing non-trivial. In higher level languages, you wouldn't have had to consider this issue, that for example, OxFFFFF1EA possibly could be interpreted as higher than 0x2AE whilst in a 2 compliment system by which 0xFFFFF1EA should be lower than 0x2AE.

Maximum	**Minimum**

If (maximumNumber<valueEntered)
{
  maximumNumber=valueEntered
}
	If (minimumNumber<valueEntered)
	{
	  minimumNumber=valueEntered
	}

Mean

Normally calculating the mean would be a simple calculation however we must take preventative measures in ARM assembly especially when working with negative numbers. We can't just simply compute:

```
mean = sumOfTotals/Count;
```

We can't do this for a number of reasons first we need to deal with the possibilities of a negative sum of totals and we are unable to divide normally like we do in a high level language.

The problem with computing negative sum of totals, is the amount of time it would take to compute. E.G. If the sum of totals = 0xFFFFFA23 across three values, we would have to keep subtracting three from 0xFFFFFA23 until it reached zero. This would take far too long to compute.

In order to reduce the processing impact of this calculation, I made improvements to this process. If the sum of the totals value is above the wrap around point of the 2 compliment system, I then convert the sum of totals back into an unsigned value, using similar pseudo code we used to convert from and to unsigned values. If the sum of the totals was negative I would print "Mean:" followed by a minus symbol. "-".

I then allowed the unsigned sum of the totals value to be used to calculate the mean, if the last value printed matched "-", I would then convert the mean into a signed value. This process is only useful for stage two, in stage three I avoid converting the mean back into a signed value as I am going to print it, which is only possible to display properly as an unsigned value.

In ARM Assembly, instead of dividing a number we have to create a while loop to subtract the count from the sum of totals until the total sum of the numbers is less than the count. This can be achieved with the following pseudocode:

```
while (total sum of numbers>= count)
{
    quotient++;
    remainder = remainder – count;
}
```

Additional Statistical Measure – Range – Extra Mile

I also managed to get my program to compute the range correctly with both positive and negative numbers. This was achieved by first converting the maximum and the minimum back into their unsigned values, this made it easier to calculate the range between two negative values. This was done using the following psuedocode, this code is quite similar to that which can be seen in stage one of the assignment however there is a lack of a Boolean here:

```
    wrapAroundPoint = 2147483647;
if (minNumber>wrapAroundPoint)    //Converts negative numbers from 2 Compliment Form
{
        value = NOT value (invert bits)
        value = value + 1 (add 1)
}
```

The same type of pseudocode is then repeated however using the maximum number. After both the maximum and minimum numbers have been converted back into their unsigned values, the calculation is quite elementary.

```
        range = maxValue - minValue;
```

Methodology for Stage Two - Sample Sequence and Statistic Computation

Input: Followed by "Enter" and "."	**Input: Hexadecimal Values**	**Output: Applicable Registry Value**	**Console Output After Welcome**	**Correct/Wrong? Why?**
"0"	0x30	Count: 0x01 Sum:0x00 Max:0x00 Min:0x00 Mean:0x00 Range:0x00	N/A – Stage 2 "Mean:"	Correct – Managed to compute all values correctly.
"230" "320" "500"	0x32,0x33,0x30 0x33,0x32,0x30 0x35,0x30x0x30	Count: 0x03 Sum:0x41A (1050) Max:0x1F4 (500) Min:0xE6 (230) Mean:0x15E (350) Range:0x10E (270)	N/A – Stage 2 "Mean:"	Correct – Managed to compute all values correctly.
"-230" "-320" "-500"	0x2D,0x32,0x33,0x30 0x2D,0x33,0x32,0x30 0x2D,0x35,0x30,0x30	Count: 0x03 Sum: 0xFFFFFBE6 (-1050 2 Compliment) Max:0xFFFFFF1A (-230 2 Compliment) Min:0xFFFFFE0C (-500 2 Compliment) Mean:0xFFFFFEA2 (-350 2 Compliment) Range:0x10E (270)	N/A – Stage 2 "Mean:-"	Correct – This program managed to handle all of these negative values correctly, it even handled the range of the negative numbers correctly.
"m"	0x6D	Count: 0x00 Sum: 0x00 Max: 0x00 Min: 0x00 Mean:0x00 Range:0x00 (No Change at all)	"Invalid Input, please restart and enter valid input"	Correct – The program handled the case of a non-number correctly.
No Input	None	Count: 0x00 Sum:0x00 Max:0x00 Min:0x00 Mean:0x00 Range:0x00 (No Change)	N/A – Stage 2	Correct – The program treated no input followed by the "enter" and "." key as if it was zero, this is the quite reasonable

				handling. And would be more than suitable given the user was meant to be “well behaved”.

3. Displaying the Mean

As I discovered during the course of this assignment, printing values is a lot trickier with ARM Assembly language in comparison to other higher level languages. In order to print the mean we must figure out the value of each separate digit first, then convert that to an ASCII symbol and then print each number symbol separately to show the full value. If this was another language like java we could print the mean using the following code:

```
        System.out.print(“”+mean);
```

But this is sadly just not possible in ARM assembly. To print the mean we must first use a powers of 10 program, and attempt to divide the mean by various powers of 10 in decreasing order.

```
//maxPowerOfBaseTen originally equals 9 but is reduced by one each time

//later in the overall loop

if (baseTenPowerCounter != maxPowerOfBaseTen)

{

        resultOfTenToThePower = 10*resultOfTenToThePower

        baseTenPowerCounter = baseTenPowerCounter++

}
```

Printing Mean - Extra Mile – Leading Zeros

Using the above program we are able to compute up to one billion. Now we can check to see is the mean greater than or equal to the result of ten to the power of X, if it isn’t the program can loop back around and try a smaller exponent, if it is we can begin to divide. By doing this comparison we are able to prevent leading zeros from being printed.

```
while(mean > = resultOfTenToThePower)

{

        mean- resultOfTenToThePower;

        valueToBePrinted++;

}
```

After we can check to see how many times the mean can be divided by a power of 10 and we have that value stored in valueToBePrinted, we then add 0x30 to that value to convert it back into an ASCII symbol and print it. Please consult the explanatory example of this process on the following page.

Worked Explanatory Example:

Mean: 1239

PowerProgram – Divide by 1,000,000,000 - Too big won't divide

PowerProgram – Divide by 100,000,000 - Too big won't divide

PowerProgram – Divide by 10,000,000 - Too big won't divide

PowerProgram – Divide by 1,000,000 - Too big won't divide

PowerProgram – Divide by 100,000 - Too big won't divide

PowerProgram – Divide by 10,000 - Too big won't divide

PowerProgram – Divide by 1000 - Will divide

1239 – 1000 = 239 valueToBePrinted = 1

valueToBePrinted + 0x30 = 0x31

Print valueToBePrinted: "1"

PowerProgram – Divide by a 100 - Will divide

239 – 100 – 100 =39 valueToBePrinted = 2

valueToBePrinted + 0x30 = 0x32

Print valueToBePrinted: "2"

PowerProgram – Divide by a 10 - Will divide

39 – 10 – 10 – 10 = 9 valueToBePrinted = 3

valueToBePrinted + 0x30 = 0x33

Print valueToBePrinted: "3"

PowerProgram – Divide by 1 – Will divide

9 -1(9) = 0 valueToBePrinted = 9

valueToBePrinted + 0x30 = 0x39

Print valueToBePrinted: "9"

Combined console output = "1239"

My brief mention of my solution to prevent leading zeros has a few exceptions that I solved using the following practices in this paragraph so to allow zeros to be printed after a significant number. At the start of this loop the valueToBePrinted is equal to zero, the end of this program checks is valueToBePrinted greater than zero (0x00) from the last loop, if so and there was no division this time around, it prints another "0" (0x30). This allows no leading zeros to be printed whilst still printing significant numbers between digits

E.G. 10010 can be printed correctly instead of printing 11, which would be incorrect.

Two Decimal Points – Extra Mile

For stage three, I also enabled by program to compute the mean in decimal form, this has enabled the mean to be more accurate however it makes it harder for my program to compute big numbers as it takes longer.

To enable this program to display decimal form I multiplied the mean by 100. I also printed a decimal point in the right point by specifying that one be printed after the powers program reached a specific count.

E.G. 3/2 = 1.5

However to work this out we can do this:

3 x 100 = 300

300/2 = 150

After we place the decimal point correctly

= 1.50

Printing Values Other Than the Mean – Extra Mile

Displaying Negative Values Correctly – Extra Mile

After I managed to print the mean correctly, I then moved on to printing over values. This process was quiet simple as I had already created the foundations to print numbers when I created a loop to print the mean. I created a stageOfPrinting function to allow the program to printed the values once and move on. After a value was printed the stageOfPrinting would increase by one.

All that was necessary for me to print additional values was to convert them from signed to unsigned, print the message describing the value E.G. "Range:" and print a "-", if necessary. I would then move the value I wanted to print into where then mean was previously stored. I would then use the extremely useful feature in ARM Assembly of branching, to branch back unconditionally into the previous printing loop. This enabled me to print any values I wanted. Despite the process of displaying other values being made easier as I had the previous printing loop ready, it is still nowhere near as simple as this process would be on a higher level language.

Methodology for Stage Three - Displaying the Mean

Input: Followed by "Enter" and "."	Input: Hexadecimal Values	Output: Applicable Registry Value	Console Output After Welcome	Correct/Wrong? Why?
"230" "320" "500"	0x32,0x33,0x30 0x33,0x32,0x30 0x35,0x30x0x30	Count: 0x03 Sum:0x41A (1050) Max:0x1F4 (500) Min:0xE6 (230) Mean:0x15E (350) Range:0x10E (270)	"Mean:350.00 Range:270.00 Sum:1050.00 Min:230.00 Max:500.00"	Correct – Managed to compute all values correctly.
"-230" "-320" "-500"	0x2D,0x32,0x33,0x30 0x2D,0x33,0x32,0x30 0x2D,0x35,0x30,0x30	Count: 0x03 Sum: 0xFFFFFBE6 (-1050 2 Compliment) Max:0xFFFFFF1A (-230 2 Compliment) Min:0xFFFFFE0C (-500 2 Compliment) Mean:0xFFFFFEA2 (-350 2 Compliment) Range:0x10E (270)	"Mean:-350.00 Range:270.00 Sum:-1050.00 Min:-500.00 Max:-230.00"	Correct – This program managed to handle all of these negative values correctly. The range was also handled correctly despite being negative.
"450780" "-450780" "90"	0x34,0x35,0x30 0x37,0x38,0x30 0x2D,0x34,0x35,0x30,0x37,0x38,0x30 0x39,0x30	Count: 0x03 Sum: 0x5A (90) Max: 0x0006E0DC (450780) Min:0xFFF91F24 (-450780 2 Compliment) Mean:0x1E (30) Range: 0xDC1B8 (901560)	"Mean:30.00 Range:901560.00 Sum:90.00 Min:-450780.00 Max:450780.00"	Correct – The program was able to compute these larger figures correctly.

“m”	0x6D	Count: 0x00 Sum: 0x00 Max: 0x00 Min: 0x00 Mean:0x00 Range:0x00 (No Change at all)	“Invalid Input, please restart and enter valid input”	Correct – The program handled the case of a non-number correctly.
“0”	0x30	Count: 0x01 Sum:0x00 Max:0x00 Min:0x00 Mean:0x00 Range:0x00	“Mean:. Range:. Sum:. Min:. Max:.”	Correct – It does process the value zero correctly here, due to my prevention of leading zeros it has caused no zero to appear here, this is the one case in which this would occur. I still believe this is acceptable output.
No Input	None	Count: 0x00 Sum:0x00 Max:0x00 Min:0x00 Mean:0x00 Range:0x00 (No Change)	“Mean:. Range:. Sum:. Min:. Max:.”	Correct – The program treated no input followed by the “enter” and “.” key as if it was zero, this is the quite reasonable handling. And would be more than suitable given the user was meant to be “well behaved”.

Breakdown of My Assignment:

Stage 1:

Can read user's number input.
Extra Mile – Can also read negative numbers correctly.
Extra Mile – Prevents users from entering in non-numbers excluding the keys user to control the program.

Stage 2:

Calculates the sum, min, max, count and mean of the data.
Extra Mile – Calculates negative numbers across all these statistical measure correctly
Extra Mile – Additional Statistical Measure calculated – Calculated the Range – Working with negative values

Stage 3:

Printing the mean to the console
Extra Mile – Calculating up to two decimal places for the mean
Extra Mile – Printing other statistical measures for display, besides the mean
Extra Mile – No leading zeros – whilst retaining significant zeros
Extra Mile – Displaying negative results correctly

All Possible Extra Miles Achieved

Limitations of Assignment:

By enabling the user to have two decimal places answers, it means this program takes slightly longer to compute big values and it puts a lower limit on the maximum value that can be entered after the mean has been multiplied by 100.

However I believe this limitation is worthwhile, as it means the user can have greater precision when working with smaller numbers, as the user can see the two decimal places instead of rounding off.